

# 猜数字游戏的计算机求解

2010011269 莫涛

(清华大学计算机系 计02班 北京 100084)

**摘要：**本文讨论了猜数字游戏的计算机求解的若干种策略，研究了如何减少平均猜测次数，优化启发式算法的运行效率，并以 C++ 高效的实现了文中提及的算法。

**关键词：**筛选法，估价函数，等价类优化

# 目录

- 1 引言
  - 1.1 标准规则
  - 1.2 变种规则
- 2 标准规则下猜数字游戏的求解策略
  - 2.1 统一化标准
  - 2.2 筛选法
  - 2.3 简单策略
  - 2.4 启发式策略
    - 2.4.1 估价函数的设计
      - 2.4.1.1 最坏可能集大小
      - 2.4.1.2 平均可能集大小
      - 2.4.1.3 预期步数
      - 2.4.1.4 反馈种类数
    - 2.4.2 算法的优化
    - 2.4.3 源码和程序展示
      - 2.4.3.1 库函数
      - 2.4.3.2 主程序
- 3 其它的尝试
  - 3.1 最优化策略
  - 3.2 数位更多的游戏
- 4 结束语
  - 4.1 总结
  - 4.2 下一步研究方向
- 5 参考文献
- 6 附录

# 1 引言

猜数字（又称 **Bulls and Cows**）是一种大概于20世纪中期兴起于英国的益智类小游戏。一般由两个人玩，也可以由一个人和电脑玩，在纸上、在网上都可以玩。这种游戏规则简单，但可以考验人的严谨和耐心。

## 1.1 标准规则

通常由两个人玩，一方出数字，一方猜。出数字的人要想好一个没有重复数字的4位数（第一位可以为0），不能让猜的人知道。猜的人就可以开始猜。每猜一个数字，出数者就要根据这个数字给出几A几B，其中A前面的数字表示位置正确的数的个数，而B前的数字表示数字正确而位置不对的数的个数。

如正确答案为 5234，而猜的人猜 5346，则是 1A2B，其中有一个5的位置对了，记为1A，而3和4这两个数字对了，而位置没对，因此记为 2B，合起来就是 1A2B。接着猜的人再根据出题者的几A几B继续猜，直到猜中（即 4A0B）为止。

猜数字游戏通常设有猜测次数的上限。根据计算机测算，如果采用严谨的猜测策略，任何数字最多7次就可猜出（即达到 4A0B）。值得注意的是，在有些地方把次数上限定义为最多几次猜测以后就可以肯定数字是几，但这时或还需要再猜一次才能得到 4A0B 的结果。

## 1.2 变种规则

标准的猜数字游戏由10个数码（0-9）和4个数位组成。可以通过变化数码或数位来丰富游戏。例如，可以使用9个数码玩4个数位的游戏。

另一种变种规则允许重复的数码。这种规则的游戏被称为 **Mastermind**。

其规则大致为：除了标准规则外，如果有出现重复的数字，则重复的数字每个也只能算一次，且以最优的结果为准。例如，如正确答案为5543，猜的人猜5255，则在这里不能认为猜测的第一个5对正确答案第二个，根据最优结果为准的原理和每个数字只能有一次的规则，两个比较后应该为1A1B，第一个5位子正确，记为1A；猜测数字中的第三个5或第四个5和答案的第二个5匹配，只能记为1B。当然，如果有猜5267中的第一个5不能与答案中的第二个5匹配，因此只能记作1A0B。

标准的 **Mastermind** 有4个数位，每个数位可以是0-5，事实上，人们进行游戏时通常用 A-F 来表示。而 **Mastermind** 还有多个变种，如 **Royale Mastermind**（5个数位，每个数位可以是0-4），**Mastermind Challenge**（双方轮流给对方猜数字），**Word Mastermind**（将数字0-25看作字母 A-Z，预想的字母串要求是一个单词），可见这类游戏在国外是非常流行的。

另一种更有挑战性的变种游戏是 **Static Game**，它要求游戏者的猜测不依赖于另一名游戏者所给的回答。即游戏者只需要一次性提出若干个猜测，然后无论另一名游戏者预想的数字是什么，根据其回答都能直接确定出答案。

如对于标准的 **Mastermind**，Greenwell 给出了一组只包含6个猜测的解：

(1, 2, 2, 1), (2, 3, 5, 4), (3, 3, 1, 1), (4, 5, 2, 4), (5, 6, 5, 6), (6, 6, 4, 3)

尽管理论上5次猜测是足够的，但至今没有找到这样的解。

## 2 标准规则下猜数字游戏的计算机求解

猜数字游戏是一个经典的信息交互的问题，游戏者需要以尽量少的询问，从另一名游戏者（或者计算机）处获取信息，从而推理出其预想的数字。

该游戏的复杂之处在于询问是分次进行的，这就意味着游戏者可以根据前面若干次询问的回答，调整自己的策略，以期达到更好的游戏效果。

本节将以分析该游戏的本质特点入手，详细讨论目前最常见的求解算法——筛选法，对不同策略的理论基础，实现，运行效果，时间复杂度等方面做一个比较，并给出了一个优化启发式算法的重要技巧，最后展示了程序代码。

### 2.1 统一化标准

首先做一些约定以方便以统一的标准来衡量不同策略的效果：

1. 在标准规则下进行，即4个数位，每个数位可以是数字0-9，各个数位的数字不一样。
2. 游戏结束条件为某次猜测的结果为4A0B，这次猜测也要计入总次数中。
3. 另一名游戏者是公正的，即他不会透漏信息或者给出错误的回答以扰乱游戏进度，也不会为了刁难游戏者而在游戏过程中更改答案。
4. 必须在7次猜测内完成任何游戏，此前提下，尽量减小平均猜测次数。这一要求将通过一一测试所有的5040种答案来统计。
5. 尽量优化程序的运行效率，每次游戏的时间至多为1秒。

对于规则2：

这个限制对于不同策略的表现有着些微的影响，即游戏者可能在不确定的情况下偶然的直接猜中答案，而也可能通过前面的猜测结果确定了答案，却不得不使用一次猜测来达到4A0B。

不过从概率的角度看，这对于询问次数的期望值不会有本质的影响。而且这个规则更加符合人的直观感觉，即以猜出答案而非知道答案作为游戏的结束，因此该规则是合理的。

对于规则3：

前一种行为是作弊没有必要考虑，但后者却是完全可以在符合游戏规则的情况下做到的，即他如果希望的话，完全可以在保证答案存在的前提下，每次给你包含信息量最少的答案。

譬如你确定了前3位是012，准备通过依次猜0123, 0124……0129来得到答案，平均来说，只需要 $7/2=3.5$ 次猜测即可完成，但他却可以在发现你这一意图时将答案

设定为0129使你不得不耗费7次猜测。

故规定另一名游戏者必须在游戏开始前确定答案，并完全按照规则进行回答，或者可以直接将其视为一个计算机程序以保证公平性。事实上，我确实编写了一个库函数 `game.h` 实现该功能，详见程序展示部分。

对于规则4:

求解猜数字游戏的策略通常有两个目标：一是保证在猜测次数限制下赢得游戏，二是使用尽量少的猜测次数。第一个目标追求的是最坏情况下的猜测次数最少，第二个目标追求的是平均情况下猜测次数最少。

对于某些数码和数位的规则组合，这两个目标不能同时实现。例如 **Mastermind** 游戏，平均猜测次数最少的策略需要平均 **4.340** 次，但最坏需要**6**次猜测；如果限制猜测次数最多为**5**次，则平均猜测次数最少的策略需要平均 **4.341** 次。

不过标准规则下，绝大部分实现了目标二的策略，最坏情况下均只需要**7**次猜测，同时目标一的优先级显然更高，因而这样的限制很有必要。

对于规则5:

从理论上说，由于每次猜测的选择是有限的，并且一定可以在有限次数内猜出所有答案。故若没有时间限制的话，计算机可以穷举所有猜测与所有回答的可能性，从中找出最佳的策略。

另一方面来说，即便在一秒的限制下，进行所有的猜测也需要**5040**秒即**1小时24**分钟的时间，不便于通过统计信息调整策略，故优化时间复杂度也可以从另一个方面提升策略的质量，是非常有价值的。

后文将提出一个对于启发式算法非常行之有效的优化，该优化将原本**3000**多秒的运行时间降到了**100**秒以内。

## 2.2 筛选法

观察上述标准规则以及变种规则下的数位与数码的范围，可以发现，无论是哪种规则下的游戏，一个共同的特点是可能的答案并不多，如标准规则下不同的答案的个数为 $10 \times 9 \times 8 \times 7 = 5040$ ，**Mastermind** 下为 $6^4 = 1296$ ，**Royale Mastermind** 下为 $5^5 = 3125$ ，**Word Mastermind** 下由于单词的限制，也不会太多。

出于游戏性的考虑，这一特点很自然，人的计算能力非常有限，即便是几千种可能性人们也不可能一一考虑到（当然，这对计算机并不困难），情况再复杂一点人们的逻辑推理能力就毫无用武之地了。

另外一个特点是每个数字的范围也不能太广，这是前一特点的直接结果，试想如果每个数位可以是**0-99**，两个数位就将有 $100^2 = 10000$ 种可能性。

还可以从更直观的角度阐述这一特点，人工求解这一类猜数字问题的一般性思路是先确定答案中有哪些数字，即尽快的得到类似**1A3B** 或者**2A2B** 的答案，再通过交换数字的顺序得到正确的答案，过多的数字不利于这一策略的实施。

利用这两个特点有两种设计算法的思路，其一是模仿人工求解的技巧，通过流程

化的猜测与逻辑推理来确定答案，但稍加尝试就可以发现这一策略不仅难以编程实现，而且效果相当不稳定。

其原因在于该思路没有利用到本质特点——“可能结果的有限性”，故下面介绍基于这一特性的筛选法，这是猜数字游戏中最常用的算法，后文的各种策略均是在这一算法的基础上实现。

该算法流程如下：

- S1. 初始化“可能集”，包括所有可能5040种答案。
- S2. 利用某种策略，进行一次猜测，得到反馈结果  $xAyB$ 。
- S3. 将可能集中所有不符合该反馈的答案删去。
- S4. 若可能集中剩余超过一个答案则返回 S2。
- S5. 剩下的唯一可能即所求答案。

相应的伪代码如下：

```
Solve()
{
    可能集 S = 所有可能的答案; (即 S={0123,0124……9876})
    While (S 的大小 > 1) {
        X = find(S);
        R = 猜测 X 的反馈;
        for 所有 S 中的元素 Y
            若答案为 Y 时猜测 X 不返回 R, 则从 S 中删除 Y
    }
    返回 S 中的唯一元素作为答案
}
```

通过每次猜测筛去不可能的答案，这也是“筛选法”这一名称的由来。

筛选法的关键步骤在于 S2，即伪代码中的 find 函数。一个优秀的策略应当充分的利用“可能集”当前的情况作为信息，设计出最合理的猜测，所谓合理也即在平均情况下尽量少的猜测完成游戏。这与人工求解时，利用已知信息进行推理是类似的。

下面介绍两类最常见的策略——简单策略与启发式策略。

## 2.3 简单策略

一个很直观的想法是应当猜测可能集中的答案，不论反馈结果如何，至少这样猜测有更大的概率命中正确答案。那么具体来说，选择可能集中的哪一个元素呢？

有两种考虑，其一是假设可能集中的元素按字典序<sup>1</sup>排列，取字典序最小的元素，这样在答案的字典序偏小时策略将有很好的表现。

相应的 find 函数设计如下：

```
find(S)
{
    将可能集按字典序从小到大排序
    返回第一个元素
}
```

该策略的表现如下：

平均猜测次数 : 5.560

最多猜测次数 : 9

总耗时 : 10.367 秒

每种猜测次数完成游戏的局数如下：

次数	1	2	3	4	5	6	7	8	9
局数	1	13	108	596	1668	1768	752	129	5

另一种考虑是随机挑选一个元素，以期使平均效果尽量好。

相应的 find 函数设计如下：

```
find(S)
{
    等概率地随机返回可能集中某个元素
}
```

该策略的表现如下：

平均猜测次数 : 5.465

最多猜测次数 : 9

总耗时 : 10.508 秒

每种猜测次数完成游戏的局数如下：

次数	1	2	3	4	5	6	7	8	9
局数	1	10	114	593	1844	1804	626	47	1

显然，后者的表现远好于前者，猜测次数超过7次的游戏局数大大减少，故随机策

<sup>1</sup> 对于两个不一样的数字串 A, B, 找出最小的 p 使得  $A_p \neq B_p$ , 若  $A_p < B_p$ , 则称  $A_i$  的字典序小于  $B_i$ , 否则反之。按这种方法比较大小进行的排序即称为字典序。

略从总体来说优于针对性过强的策略。尽管如此，它们仍均不满足2.1中的规则4，最坏情况下9次猜测才完成游戏。不过其运行时间均只有10秒左右，非常优秀。

## 2.4 启发式策略

简单策略之所以效果不佳，问题在于猜测的数字任意性太大，不妨考虑枚举所有的数字串作为猜测，从中挑选“最佳”作为下一次猜测。

这一思路有两个问题：

其一，“最佳”如何定义？枚举剩下的游戏过程的所有可能性选出猜测次数最少的自然是最好的，但由于时间的限制无法实现。

这里提出估价函数的概念：利用有限的信息，根据求解问题的特点，判断一个行动的优劣程度，并进一步地将优劣程度以一个数字表现出来。

譬如在猜数字游戏中，一个容易想到的估计函数是可能集的大小，即在这次猜测后，剩余的可能性应尽量少，因为大致说来，可能集越小所需的猜测次数越少。

估价函数是已知信息的一种高度概括，它并不完全精确，但在很大的程度上体现了人们的主观性，因而使用了估价函数的程序往往看上去非常智能。譬如著名的国际象棋电脑深蓝，使用的便是极大极小搜索+估价函数的算法，其设计过程中曾聘请大批的国际象棋专家设计与调整估计函数。

其二，枚举的数字串是否应局限于可能集中的元素？从简单策略的角度来看，这一想法是理所当然的。但理性地看，就获得信息来说，那些非答案串的契合度可能更好，而且，如果估价函数设计得足够合理，更广的枚举面自然会有更好的效果。

综合上述两点，启发式算法的 find 函数如下设计<sup>2</sup>：

```
find(S)
{
    ret = 0123
    for 所有可能的答案（5040种）X
        若 value(X)<value(ret)则 ret=X
        若 value(X)=value(ret)且 X 在 S 中 则 ret=X
    返回 ret
}
```

value(X)即为估价函数，这里规定，value(X)的值越小，则猜测 X 的效果就更好，最佳猜测应为所有 X 使 value 取到最小值的那一个。

估价函数是启发式算法中最重要的部分，算法的时间效率与表现相应地取决于估价函数的计算复杂度与合理程度，下面进行详细讨论。

<sup>2</sup> 对于问题二，这里使用了折中的办法，枚举所有可能的答案，当估价函数相等时优先可能集中的元素



## 2.4.1 估价函数的设计

### 2.4.1.1 最坏可能集大小

估价函数的依据只有当前的可能集  $S$ ，以及待估价的猜测  $X$ 。假设当前猜  $X$ ，则得到的反馈  $R$  有 15 种可能，分别是：

0A0B, 0A1B, 0A2B, 0A3B, 0A4B

1A0B, 1A1B, 1A2B, 1A3B

2A0B, 2A1B, 2A2B

3A0B, 3A1B

4A0B

相应的，可以计算对于得到每种反馈后剩余的可能集，故定义

$S[a][b]$  = 猜  $X$  反馈结果为  $aAbB$  时剩余的可能集

譬如估价函数：剩余可能集最坏情况下的大小，则可以设计为

value( $X$ )

{

  计算  $S[a][b]$

  ret = 0

  for a = 0 to 4

    for b = 0 to 4-a

      若  $|S[a][b]| > ret$  则 ret =  $|S[a][b]|$

  返回 ret

}

该策略的表现如下：

平均猜测次数 : 5.389

最多猜测次数 : 7

总耗时 : 79.620 秒

每种猜测次数完成游戏的局数如下：

次数	1	2	3	4	5	6	7	8	9
局数	1	3	44	519	2106	2152	215	0	0

最多只用了 7 次猜测，符合了 2.1 中的规则 4。平均猜测次数从 5.465 次降低到了

5.389次，减少了近0.1，这意味着在5040局游戏中，平均10局里有一局游戏该策略比简单策略少用一次猜测，相较来说是很大的提高。

这也从另一个角度反映了猜数字游戏的困难性，在一个正常算法的基础上，企图进行些许的提高也是较困难的。

总耗时增加到了约80秒，这是可以理解的，虽然计算最大值可以  $O(1)$  的完成，但计算  $S[a][b]$  的工作量并不小。

最朴素的实现，枚举每一个  $X$ ，扫描  $S$  中每个元素，计算其反馈  $R$ ，将  $X$  分类到某个  $S[a][b]$ ，一次决策的时间复杂度为  $O(5040 * |S| * |R|)$ 。故测试所有游戏的复杂度约为  $|S| * 5.465 * O(5040 * |S| * |R|) = O(|S|^3)$ ， $|S|$  最大时有5040，并且该算法常数非常大。

事实上，直接这样计算所需要的时间超过一个小时，上文提到的80秒是一系列优化之后的结果，这部分优化的技巧将在2.4.2节详细阐述，2.4.1节中需要计算  $S[a][b]$  的部分均默认使用了所有优化。

### 2.4.1.2 平均可能集大小

尝试过最坏情况的可能集大小，那么考虑平均情况的可能集大小效果会如何呢？

举一个只包含两个数字的例子，某时刻  $S = \{12, 21, 34\}$

1. 若此时猜  $X = 34$ ，当答案为34时反馈2A0B，当答案为12或21时反馈0A0B。

那么平均剩余  $1 \times \frac{1}{3} + 2 \times \frac{2}{3} = \frac{5}{3}$  个数。

2. 若此时猜  $X = 12$ ，当答案为34时反馈0A0B，当答案为12时反馈2A0B，当答

案为21时反馈0A2B，那么平均剩余  $1 \times \frac{1}{3} + 1 \times \frac{1}{3} + 1 \times \frac{1}{3} = 1$  个数。

显然此时猜12效果比猜34要好。

该估价函数可以设计为：

```
value(X)
{
    计算 S[a][b]
    ret = 0
    for a = 0 to 4
        for b = 0 to 4-a
            ret = ret + |S[a][b]|2
    返回 ret
}
```

**ret** 最后应除一个分母  $|S|$ ，但由于当次猜测中  $|S|$  不变，即分母相同，故只需对不同的  $X$  比较分子即可。

该策略的表现如下：

平均猜测次数：5.268

最多猜测次数：7

总耗时：73.762 秒

每种猜测次数完成游戏的局数如下：

次数	1	2	3	4	5	6	7	8	9
局数	1	4	59	573	2430	1886	87	0	0

平均猜测次数再次有了较大的提升，其余数据没有太大变化。也就是说，这一估价函数更加贴合实际的优劣程度。

### 2.4.1.3 预期步数

另一个最科学的思路是将估价函数取为所需猜测次数的期望值，即每个  $S[a][b]$  所需猜测次数乘以其出现概率的和，可以设计为：

```

value(X)
{
    计算 S[a][b]
    ret = 0
    for a = 0 to 4
        for b = 0 to 4-a
            ret = ret + |S[a][b]| * Step(S[a][b])
    返回 ret
}

```

其中  $Step(S[a][b])$  表示从该可能集中猜出答案所需要的期望步数，由于时间限制  $Step$  函数无法精确计算，只能进行大致的估计。考虑可能集每次都是缩小到原来的一部分，可以认为平均缩小为原来的  $k$  分之一，那么有

$$Step(S[a][b]) = \log_k(|S[a][b]|)$$

进一步可以发现  $k$  的具体值对于  $Step$  函数的相对大小没有影响，故可以直接取

$$Step(S[a][b]) = \ln(|S[a][b]|)$$

该策略的表现如下：

平均猜测次数：5.265

最多猜测次数 : 7

总耗时 : 95.682 秒

每种猜测次数完成游戏的局数如下:

次数	1	2	3	4	5	6	7	8	9
局数	1	4	53	560	2515	1796	111	0	0

相对前一个估价函数没有本质变化,但由于可以任意设计 Step 函数,这个思路应该是最有可扩展性,在最优化策略中将有进一步探讨。

#### 2.4.1.4 反馈种类数

这是由 Kooi 于 2005 年提出的新策略,它基于这样一种想法,较小的可能集更加有利于后续的猜测,但其成为该次猜测后的剩余可能集的概率也相对较小,即可能集的大小对于期望收益并没有影响,可以认为都是 1。

于是对于一个猜测 X,猜 X 后可能得到的不同可能集越多,其效果就越好。换句话说,应最大化不同的反馈信息的个数,该估价函数可以设计为

```

value(X)
{
    计算 S[a][b]
    ret = 0
    for a = 0 to 4
        for b = 0 to 4-a
            if (S[a][b]非空) ret = ret + 1
    返回 ret 的相反数
}

```

注意 value 是要取最小化的,而 ret 要最大化,故应返回其相反数。

该策略的表现如下:

平均猜测次数 : 5.308

最多猜测次数 : 8

总耗时 : 78.632 秒

每种猜测次数完成游戏的局数如下:

次数	1	2	3	4	5	6	7	8	9
局数	1	11	80	556	2277	1929	183	3	0

有 3 局游戏需要 8 次猜测,平均猜测次数也偏高,表现不佳。但在 Mastermind

中，这是目前已知的表现最好的策略之一，可以看到，同样的算法在不同的游戏规则下可以有很大的差异。

## 2.4.2 算法的优化

回到2.4.1.1中提出的问题， $S[a][b]$ 的计算。

分析原有的朴素算法，枚举每一个  $X$ ，扫描  $S$  中每个元素，计算其反馈  $R$ ，将  $X$  分类到某个  $S[a][b]$ 。算法复杂度似乎无法降低，但可以观察到，在一次游戏中，主要的时间消耗在前几步猜测，此时  $S$  非常大，每次扫描都将花费很多时间，而在游戏后半部分， $S$  的大小基本可以认为是常数级别。

譬如这是使用预期步数作为估价函数的启发式策略在猜测6789的过程：

答案		$ S $	本质不同 $X$ 的个数
第一次猜测	: 0 1 2 3 0A0B	5040	1
第二次猜测	: 4 5 6 7 0A2B	360	209
第三次猜测	: 8 9 7 5 1A2B	84	3360
第四次猜测	: 7 9 4 8 0A3B	21	5040
第五次猜测	: 9 8 7 6 4A0B	5	5040

而游戏前期的一个特点在于很多猜测本质上是相同的，如第一次猜测时任何  $X$  完全没有区别，可以直接猜0 1 2 3，这对于平均猜测次数没有任何影响。第二次猜测时，由于数字4 5 6 7 8 9均未使用过，故0 1 4 2与0 1 7 2也没有区别。

综合以上考虑，可以在枚举  $X$  时要求未使用过的数字从小到大使用，即若数字  $P$  从未出现在之前的猜测中，那么除非  $P-1$  出现在了以前的猜测中，或者  $P-1$  出现在了当前的  $X$  中，才允许在  $X$  中包含  $P$ ，这样恰好可以将本质不同的  $X$  各枚举一次。

加入了该优化的枚举  $X$  的伪代码如下：

```

Dfs(i)
{
    if (i = 4) 得到了一个本质不同的 X else
        枚举数字 P=0-9作为 X 的第 i 位
            若符合关于 P 的要求则 Dfs(i+1)
}

```

该优化的效果极其明显，如上例中，第三次猜测达到了最大计算量，

$$|S| * \text{本质不同的 } X \text{ 个数} = 84 * 3360 = 282240$$

远远小于原复杂度  $|S|^2 = 25401600$ 。

另外一个常数的优化在于计算反馈  $R$ ，若先枚举  $X$ ，则对于  $S$  中每个元素，为了

统计 **B** (位置不正确的数的个数), 都需重新记录其出现过的数字, 用一个 **bool** 数组的话需要**10**单位的时间。

但若先保存好所有本质不同的 **X**, 扫描 **S** 中每个元素, 对每个 **X** 计算其 **R** 值, 也即将关于 **S** 的循环放到外围, 可以省去多次的费操作, 也有一定的优化效果。

此外在实现程序时的一些细节对程序的效率也有不小的影响, 这部分过于繁杂, 这里不展开讨论。

添加了上述若干优化后, 程序的运行时间可以缩短到**100**秒左右。

其实本质上启发式策略的筛选法就是一个深度为**1**的搜索过程, 搜索中常用的剪枝技巧都可以应用到优化该算法。

### 2.4.3 源代码和程序展示

本节提到到所有程序都在 Win7 Devcpp4.9.9.2下以 C++语言开发实现。

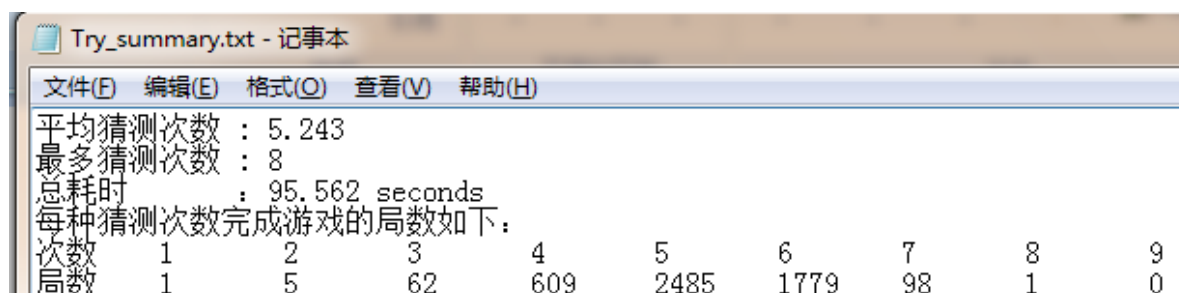
上述所有程序的运行时间均在个人电脑上统计, 本机配置为: 内存**2.00 GB**, Intel®Core™i3 CPU, 主频**2.53GHz**。

#### 2.4.3.1 库函数

使用 **Class** 实现, 禁止了主程序直接访问内部变量以保证游戏过程的公平性。库函数的使用方法如下:

1. 将代码拷贝到 **game.h** 中, 在主程序的头文件中加入 **#include <game.h>**。
2. 通过更改 **AIname** 改变输出文件名。
3. 调用 **guess(a)**可以进行猜数字游戏, 其中 **a** 为包含**4**个数字的数组或 **vector**, 表示所猜的数字, 函数以一个 **pair<int,int>**返回反馈信息。每次猜到**4A0B**后自动开始下一局游戏, 若**10**次没有猜出来该局游戏也会结束并记失败。猜完所有的**5040**局游戏后库函数将自动结束程序, 并在 **AIname\_log.txt** 中输出所有的游戏过程, 在 **AIname\_summary** 中输出总的统计信息。

下图2.4.3.2中的主程序的运行结果。



```
Try_summary.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
平均猜测次数 : 5.243
最多猜测次数 : 8
总耗时 : 95.562 seconds
每种猜测次数完成游戏的局数如下:
次数 1 2 3 4 5 6 7 8 9
局数 1 5 62 609 2485 1779 98 1 0
```

源代码如下:

```
#include <iostream>
#include <string>
```

```
#include <vector>
using namespace std;

const int L = 4, S = 10, maxt = 9, Size = 5040;
//数字个数, 数字范围[0,S), 最多猜测次数, 不同的答案个数
const pair<int, int> error = make_pair(-1, -1);
//错误猜测时范围的信息
const pair<int, int> over = make_pair(-2, -2);
//猜测次数已达最多猜测次数且未猜出答案, 当局游戏自动结束

string AIname = "Noname";

class game {
    int P, Ps, T[maxt + 1];
    //已猜局数, 总局数, T[i]表示 i 次猜出的局数, T[0]表示未猜出的局数
    int a[L], t;
    //当前局的数字, 当前局已猜次数
    int time;
    //记录时间
public:
    game()
    //初始化
    {
        P = t = 0;
        Ps = 1;
        for (int i = 0; i < L; ++i)
            Ps *= S - i;
        memset(T, 0, sizeof(T));
        for (int i = 0; i < L; ++i)
            a[i] = i;
    }
};
```

```
}  
pair<int, int> guess(int* b)  
//猜数字， b 数组为所猜的 L 个数字， 以 pair<A,B> 返回猜测结果  
{  
    if (!P && !t)  
        freopen((AIname + "_log.txt").c_str(), "w", stdout);  
    if (!t) {  
        cout << "Game  " << P + 1 << " : ";  
        for (int i = 0; i < L; ++i)  
            cout << a[i] << " ";  
        cout << endl;  
    }  
    ++t;  
    int A = 0, B = 0;  
    for (int i = 0; i < L; ++i)  
        if (b[i] < 0 || b[i] > S) return error;  
    for (int i = 0; i < L; ++i)  
        for (int j = i + 1; j < L; ++j)  
            if (b[i] == b[j]) return error;  
    for (int i = 0; i < L; ++i)  
        if (a[i] == b[i]) ++A;  
    for (int i = 0; i < L; ++i)  
        for (int j = 0; j < L; ++j)  
            if (a[i] == b[j]) ++B;  
    cout << "guess " << t << " : ";  
    for (int i = 0; i < L; ++i)  
        cout << b[i] << " ";  
    cout << " " << A << "A" << B - A << "B" << endl;  
    if (A == L || t == maxt) {  
        if (A == L) {
```



```
        ++T[t];
        cout << "Seccessful game in " << t << " guesses" <<
endl << endl;
    }else {
        ++T[0];
        cout << "Failed game" << endl << endl;
    }
    if (++P == Ps) {
        freopen((AIname + "_summary.txt").c_str(), "w",
stdout);
        //cout << P << " Games: " << P - T[0] << " Y / " <<
T[0] << " N" << endl;
        int max = 0, sum = 0;
        for (int i = 1; i <= maxt; ++i) {
            if (T[i]) max = i;
            sum += i * T[i];
        }
        cout << fixed;
        cout.precision(3);
        cout << "平均猜测次数 : " << double(sum) / (P - T[0])
<< endl;
        cout << "最多猜测次数 : " << max << endl;
        cout << "总耗时      : " << clock() / 1000.0 << "
seconds" << endl;
        cout << "每种猜测次数完成游戏的局数如下: " << endl;
        cout << "次数" << "\t";
        for (int i = 1; i <= maxt; ++i)
            cout << i << "\t";
        cout << endl;
        cout << "局数" << "\t";
        for (int i = 1; i <= maxt; ++i)
```

```
        cout << T[i] << "\t";
    cout << endl;
    exit(0);
}else {
    t = 0;
    bool u[S];
    memset(u, 0, sizeof(u));
    int x = P;
    for (int i = 0; i < L; ++i) {
        int y = x;
        for (int j = i + 1; j < L; ++j)
            y /= S - j;
        for (int j = 0; j < S; ++j)
            if (!u[j]) {
                --y;
                if (y < 0) {
                    u[j] = true;
                    a[i] = j;
                    break;
                }
            }
        y = 1;
        for (int j = i + 1; j < L; ++j)
            y *= S - j;
        x %= y;
    }
}
return make_pair(A, B - A);
}
```

```
} G;

pair<int, int> guess(int* b)
{
    return G.guess(b);
}

pair<int, int> guess(vector<int> vb)
{
    int b[L];
    for (int i = 0; i < L; ++i)
        b[i] = vb[i];
    return G.guess(b);
}
```

### 2.4.3.2 主程序

下述为以  $\ln(|S[a][b]|+1)$  作为估价函数的启发式策略的源代码，加入了上文提到的所有优化，使用 `vector` 保存数字。

代码中用注释保留了其余估价函数的计算方式，稍加更改即可测验其余估价函数的效果，简单策略的代码只需替换 `find` 函数。

```
#include <iostream>
#include <string>
#include <vector>
#include <cmath>
#include "game.h"
using namespace std;

vector<int> a;
bool u[S];
//临时变量
vector< vector<int> > f, g, h;
```

```
//所有可能的答案，当前可能的答案，临时变量
int t, ts, p;
//当前局猜测步数，总步数，已使用的最大数字
int ct[Size + 1][L + 1][L + 1];
//ct[i][j][k]表示猜测 h[i]返回 jAkB 时可能集的大小
```

```
void dfs(int i)
//扩展 f
{
    if (i == L) f.push_back(a);
    else
        for (int j = 0; j < S; ++j)
            if (!u[j]) {
                u[j] = true;
                a.push_back(j);
                dfs(i + 1);
                u[j] = false;
                a.erase(a.begin() + i);
            }
}
```

```
void dfs2(int i)
//扩展本质不同集合
{
    if (i == L) h.push_back(a);
    else
        for (int j = 0; j < S; ++j)
            if (!u[j] && (j <= p + 1 || u[j - 1])) {
                u[j] = true;
                a.push_back(j);
            }
}
```

```
        dfs2(i + 1);
        u[j] = false;
        a.erase(a.begin() + i);
    }
}

pair<int, int> calc(const vector<int> &a, const vector<int> &b)
//计算答案为 a, 猜测为 b 时的返回值
{
    int A = 0, B = 0;
    memset(u, 0, sizeof(u));
    for (int i = 0; i < L; ++i)
        u[a[i]] = true;
    for (int i = 0; i < L; ++i)
        if (a[i] == b[i]) ++A;
        else
            if (u[b[i]]) ++B;
    return make_pair(A, B);
}

vector<int> find()
//从 g 中计算出当前应该猜测的数字
{
    if (t == 1) return f[0];
    if (g.size() == 1) return g[0];
    if (p >= S - 2) h = f;
    else {
        h.clear();
        a.clear();
        memset(u, 0, sizeof(u));
    }
}
```

```
    dfs2(0);
}
//优化后的 R 值的计算,
//此时为本质不同的 X 的集合
memset(ct, 0, h.size() * (L + 1) * (L + 1) * sizeof(int));
for (int i = 0; i < g.size(); ++i) {
    memset(u, 0, sizeof(u));
    for (int j = 0; j < L; ++j)
        u[g[i][j]] = true;
    for (int j = 0; j < h.size(); ++j) {
        int A = 0, B = 0;
        for (int k = 0; k < L; ++k)
            if (h[j][k] == g[i][k]) ++A;
            else
                if (u[h[j][k]]) ++B;
            ++ct[j][A][B];
    }
}
}
double bv = 1e9;
vector<int> ret;
for (int i = 0; i < h.size(); ++i) {
    double v = 0;
    //估价函数的计算
    for (int j = 0; j < L; ++j)
        for (int k = 0; j + k <= L; ++k)
            v += ct[i][j][k] * log(ct[i][j][k] + 1);
            //v += ct[i][j][k] * ct[i][j][k]; //平均可能集大小
            //if (ct[i][j][k]) v += ct[i][j][k] * log(ct[i][j][k]); //预
期步数

            //if (ct[i][j][k] > v) v = ct[i][j][k]; //最坏可能集大小
```

```
        //if (ct[i][j][k]) --v;//反馈个数
        if (v < bv || (v == bv && binary_search(g.begin(), g.end(),
h[i]))) {
            bv = v;
            ret = h[i];
        }
    }
    return ret;
}
```

```
int main()
{
    AIname = "Expect";
    dfs(0);
    for (int P = 0; P < f.size(); ++P) {
        g = f;
        t = 0;
        p = L - 1;
        memset(u, 0, sizeof(u));
        do {
            ++t;
            a = find();
            pair<int, int> ret = guess(a);
            if (ret.first == 4) break;
            for (int i = 0; i < L; ++i)
                if (a[i] > p) p = a[i];
            h.clear();
            for (int i = 0; i < g.size(); ++i)
                if (calc(g[i], a) == ret) h.push_back(g[i]);
            g = h;
        } while (t < 1000000);
    }
}
```

```
    }while (1);  
    ts += t;  
    cerr << P << " " << t << " " << double(ts) / (P + 1) << endl;  
}  
  
return 0;  
}
```

### 3 其它的尝试

撰写本文的过程中，由于启发式算法的平均效率基本达到了瓶颈，本人进一步思考了如何从本质上改进策略以大幅优化策略额表现，取得了一些不成熟的成果。

此外在 Winter Camp 2006 guess<sup>3</sup>一题的基础上，尝试了某些非标准规则的猜数字游戏。

#### 3.1 最优化策略

虽然完整的最优化策略在实际计算中无法得到，但对预期步数这一估价函数的优化应该是可以相当接近最优化策略的。

考虑2.4.1.3节中的 Step 函数，取  $\ln(|S[a][b]|)$  在  $S[a][b]$  集合较大的时候是相当贴近实际情况的，但当  $S[a][b]$  较小时， $S[a][b]$  集合内的具体元素分布对于所需的猜测次数会有极大的影响，造成误差会使该策略的表现变差。

不过可以发现，由于  $S[a][b]$  较小，直接枚举以后所有的可能性，加上优秀的剪枝是有希望计算出  $\text{Step}(S[a][b])$  的精确值，但这个算法的编程难度过高，因而我没有编程实现。

另外一个想法是更加精确的关于  $|S[a][b]|$  的 Step 函数，譬如以  $\ln(|S[a][b]|)$ ， $|S[a][b]|^k$  等函数为基础进行计算，统计实际运行时各种大小的可能集所需的猜测次数，构造出相应的 Step 函数，并将该函数加入基础函数中，继续重复上述计算步骤，直至得到优秀的结果。这便是经典的遗传算法的思想。

我实现了这一算法的单次遗传程序（即迭代，每次从上一个函数构造下一个 Step 函数），发现实际的效果不是十分的好，耗时3个小时，迭代了100多次，只得到了一个平均猜测次数为5.250的 Step 函数。

但在调整这一算法的过程中，我摸索出了一个 Step 函数  $\ln(|S[a][b]|+1)$ ，平均猜测次数为5.243，比上述所有策略都要优。可能的原因是在  $S[a][b]$  较小时该函数的匹配程度更加精确。

---

<sup>3</sup> 这是国内水平最高的程序设计竞赛之一



## 3.2 数位更多的游戏

在数位变得更多时，可能的答案数目将极大的增加。如标准规则下，10个数位，每个数字0-9，有 $10! = 3628800$ 种可能。S[a][b]的计算将是不可能的任务，从而上述启发式策略都将无法实现。

应对这种情况的一个很好的思路是分类处理，在可能集较大时，使用简单策略，当可能集减小到足够小时，使用启发式策略甚至最优化策略。由于可能集较大时，本质不同的猜测相当少，可行选择不多，猜测的质量也不会有太大差别。而从前文可以看到，这样可以极大的节省运行时间。

## 4 结束语

### 4.1 总结

本文的主要内容是猜数字游戏最常用的计算机求解算法——筛选法，研究了包括简单策略，启发式策略，最优化策略在内的若干种策略的理论基础，编程实现，运行效果，时间复杂度。

下表为各策略的运行效果的一些关键数据。

		最多猜测次数	平均猜测次数	运行时间 <sup>4</sup>
简单	字典序最小	9	5.560	10.367秒
	随机	9	5.465	10.508秒
启发式策略	最坏可能集大小	7	5.389	79.620秒
	平均可能集大小	7	5.268	73.762秒
	预期步数	7	5.265	95.682秒
	反馈种类数	8	5.308	78.632秒
综合	遗传算法	7	5.250	97.130
	优化的 Step 函数	7	5.243	95.562

纵向比较这些策略可以发现，启发式策略是综合表现最为优异的算法。

<sup>4</sup> 其中综合策略的运行时间不包括通过遗传算法或者人工调整以确定参数所耗费的时间，而事实上，这部分工作耗费了我数个小时

该策略最大的优势在于估价函数的引入大大增加了算法的智能性，形象的说，让计算机得到了部分的推理的能力，这恰恰是人脑思考问题的最大优势，结合计算机超强的计算能力，使得程序能以较短的时间求出很优秀的解。

此外，在2.4.2节与3.2节中优化  $S[a][b]$  的计算时，都用到了一个很重要的技巧——分情况讨论，虽然没有降低算法复杂度，但极大改善了程序的运行效果。

总的来说，猜数字游戏的计算机求解实质上可以理解为人工智能问题，研究如何使计算机具备更强的推理能力是非常有价值的。

## 4.2 下一步研究方向

标准规则下的猜数字游戏还有不小的提升空间，部分最优化策略，或者完整的遗传算法，都有希望从本质上优化平均猜测次数。而这些策略在其它规则下的表现也是值得关注的问题，尤其是 **Mastermind** 的若干变种游戏。

此外，2.1节中的规则三所指出的扰乱游戏的策略的实现也是一个非常有趣的博弈论问题，该问题甚至比求解猜数字问题本身更加困难。

## 5 参考文献

[1] Barteld Kooi, University of Groningen, The Netherlands. (2005) Yet Another Mastermind Strategy

[2] Donald, Knuth. Stanford (1976) The Computer As Master Mind.

[3] 猜数字的一种解法.

<http://www.cppblog.com/lemene/archive/2007/11/26/37273.html>

[4] Mastermind – Wikipedia.

[http://en.wikipedia.org/wiki/Mastermind\\_\(board\\_game\)](http://en.wikipedia.org/wiki/Mastermind_(board_game))

[5] 猜数字 – 百度百科.

<http://baike.baidu.com/view/358630.htm>

[6] Solution for guess, Winter Camp 2006

## 6 附录

三个最喜欢的老师：

1. 孙茂松教授（计算机科学与技术概述）
2. 孙家广院士（信息科学与技术的发展）

### 3. 张钹院士（计算机的昨天今天明天）

改进意见：

1. 希望讲课的内容更加系统化，能让我们对信息科学与技术的整体构架有一个更加清晰地认识。
2. 以小组形式完成课程论文，本次独立研究猜数字问题感觉困难不小，有不少好想法但无力全部实现，只得到了一些初步成果。而且我认为交流能力与团队合作能力在科研中有非常重要的地位。

祝信概课越办越好！